A Multipath Routing Algorithm for Encoding and Decoding Linear Locations in TPEG2-ULR

Progress Report, TPEG-LOC2 Study

Fraunhofer FOKUS
February 26, 2013

Authors:
Thilo Ernst
Hans-Werner Pohl
Birgit Kwella
Matthias Schmidt

Table of Content

Introduction 3
Linear Locations and how they are transmitted 4
Segment matching 4
Map matching process for linear locations 6
Encoding a Linear Location 11
Conclusion 15
References 15

Introduction

TPEG (Transport Protocol Experts Group) [1] is a protocol specification for the exchange of RTTI (Real Time Traffic Information) to provide road users with comprehensive, up-to-date traffic and traveller information across multiple transport modes, thus allowing door-to-door support. TPEG consists of a collection of ISO/TS standards that already cover a number of essential applications, and is extensible through new applications. It is being constantly developed further by the TISA (Traveller Information Services Association) [2]. TISA is a non-profit organization with more than 100 members from industry and public institutions.

TPEG has the aim to describe real-world events and situational data with their spatio-temporal scope, i.e. the geographical location and time or time period they are associated with. Those (event and situational data) originate in established sensor, control and data management infrastructures e.g. for municipal traffic management, public transport operation and weather observation. Some of these data is currently distributed to vehicle-based and other mobile client devices through the widely-deployed RDS-TMC system [4]. However, this is only a small subset of the RTTI information available today because of severe limitations of TMC [3], which in turn resulted from the very low transport capacity of its original carrier RDS. To overcome these limitations TPEG was created. Beside the traditional but limited scope of TMC, TPEG addresses new application areas, e.g. public transport, weather, current traffic flow and prediction, parking information (and more) in the form of TPEG application specifications. Furthermore it remains extensible through additional future applications and a process for their specification within the TISA and their representation in the form of standards or specifications [6][7]. All TPEG applications need provisions to locate the event or situational data they express precisely in their corresponding geographical region.

In the TPEG specification three major containers hold all necessary information to manage the information flow of RTTI messages (that express events and situational data) from producer side to consumer[1] side. The management container holds all essential temporal and administrative information to manage each message during its lifetime. The event container holds all information to describe the event itself and further details of it. The location referencing container holds all spatial information to describe the geographical location and scope of this event; more concretely, it may hold one or several location references (all of which describe the same conceptual location, only using different location referencing methods). The TPEG protocol structure is defined in a way that a client may concentrate on certain applications and containers and can easily skip messages which are of no interest to the client. Unknown elements in the input stream (e.g. messages belonging to TPEG applications that were not yet defined or were disregarded at the time the client software was implemented) will be skipped as well; the same holds true for the different types of location references.

Several types of location references (and the location referencing methods used to encode and decode them) already are specified as standards or as drafts in TPEG[2].

[1] in TPEG *producer* is used synonymously to *server* and *consumer* synonymously to *client* as the standard usage scenario is infrastructure-to-vehicle

[2] Apart from the TPEG-LOC format to be superseded by ULR, the most important types are TMC/ALERT-C location references and DLR1 (AGORA-C) location references. Recently another open alternative, OpenLR, has been introduced to the TPEG standardization.

TPEG2-ULR (Universal Location Referencing) is a new location referencing method and type developed by Fraunhofer FOKUS (earlier: Fraunhofer FIRST) in close coordination with TISA, public broadcasters and industrial partners, which aims to overcome the limits of TPEG-LOC [5] (in terms of efficiency and accuracy) based on an open, royalty-free method. Another essential goal of ULR is to provide a location referencing scheme that is both human-readable and machine-processable, and supports basic functionality even if the client system does not have an on-board digital map. An ULR location reference contains geo-graphical co-ordinates according to the WGS84 standard and additional information such as direction and height or level if applicable. The goal is to always provide enough information to have realistic chances to reconstruct the location with sufficient accuracy on the consumer side so the user or system on the client-side can correctly react on the message. Some parameters are mandatory (i.e. will always be included by the producer) and some others are optional. The consumer side can always rely on the mandatory part and can profit from the optional parameters if they are present and the client system is capable of processing them.

TPEG2-ULR was introduced in [8] and [9]; the focus of these papers was the disambiguation of individual road segments through the use of special matching information, including recurrence values computed form a Markov-chain based synthetic flow model operating in a circular vicinity around the transmitted segment identification point.

An important use case for a location reference method is map matching for linear (line-shaped) locations. This publication focuses on how map matching for linear locations works in ULR, and specifically introduces a routing algorithm designed for this purpose.

Linear Locations and how they are transmitted

A linear location is a location that can be represented as an (acyclic and non self-crossing) path drawn over the source map's road network. In ULR, a linear location is transmitted as an ordered sequence of Segment identification points (SIPs), to each of which matching information (supporting the correct choice of one or more matching segments from the target map) may be attached. As map matching for linear locations (like map matching for point locations) in ULR builds upon matching functionality for single segments, the next section provides more details about the segment matching process than covered in earlier publications. The remaining sections describe how map matching works when decoding ULR linear locations, and how linear locations are encoded. The focus of the paper is on the high-level processing methods so protocol-level encoding specifics are not covered here in any detail.

Segment matching

Map matching in ULR is segment-centric. The elementary unit of map matching compares a transmitted segment identification point (SIP) E to a set of consumer-map segments S (e.g. all segments in a certain radius around E) and involves these steps:

1. determination of a consumer map segment s from S, best matching a transmitted SIP, and
2. determination of a map-matched point E' on segment s.

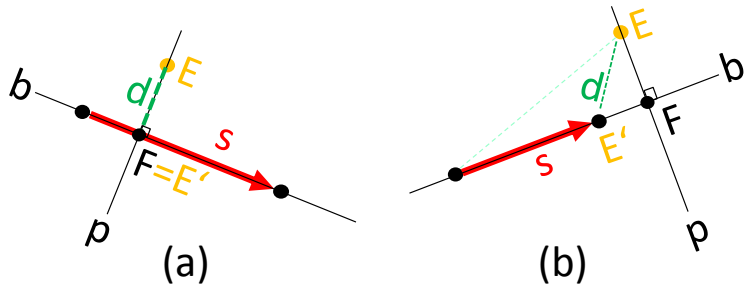

Figure 1: Minimal point-segment distance and optimum on-segment point position

Minimal point-segment distance and optimum on-segment point position

Step 1 is done by computing a penalty function prominently involving the distance between E and s, for all s in S, and determining the minimum. The following method for computing the distance between point E and segment s (also referred to as **mindist()** function subsequently) is used (see Figure 1): Construct a line b which goes through both end points of s. Construct a line p which is perpendicular to b and goes through point E. The intersection of b and p is the "foot point" F. If F is situated between both end-points of s (case (a) in illustration), the result d is the distance between E and F. Else (if the foot point is situated outside s – case (b) in illustration), the result d is the distance between E and the closest end point of s.

Step 2 is carried out after a segment has been selected as the best match, in order to determine a concrete position on the segment which most closely matches the transmitted identification point (optimum on-segment point position, also referred to as function **oospp()** subsequently). Concretely, in case (a), the foot point F will be used as E', in case (b), E' will be placed on the closest end-point of s.

Individual segment matching

Given a transmitted SIP P with attached matching information, the following steps are taken:

1. Enumerate those segments s from the consumer-side map which are situated in a circular region of interest (ROI) with radius R around P, or which intersect this ROI
2. Compute the Markov recurrency values for all segments within the ROI
3. For each segment, compute a matching penalty value MP(s) as a weighted function adding
    - the segment's distance from P: mindist(s, P)
    - a "length-weighted absolute heading difference" lw_abs_heading_diff that is computed from the consumer-map segment's heading value s.heading, the segment's metric length s.length, and the matchInfo attributes of P according to the following algorithm:

        - length_weight := s.length / 50.0
          if (length_weight >1.0): length_weight := 1.0
          if (length_weight < 0.1): length_weight := 0.1

- **if** (matchInfo.roadAttrs.oneWay and s.oneWay) :
  abs_heading_diff **:= abs**(matchInfo.heading – s.heading)
  if (abs_heading_diff **>** PI): abs_heading_diff **:=** 2***PI** - abs_heading_diff
  lw_abs_heading_diff **:=** length_weight * abs_heading_diff
- **else:**
  abs_heading_diff **:= abs**((matchInfo.heading **mod PI**) - (s.heading **mod PI**))
  if (abs_heading_diff **> PI** / 2): abs_heading_diff **:= PI** - abs_heading_dif
  lw_abs_heading_diff **:=** length_weight * abs_heading_diff

- the absolute value of the difference between the segment's normalized road class value and the transmitted normalized (to range 0..1) road class value: abs(s.category - matchInfo.roadAttrs.category)
- the sum of absolute values of the difference between received and locally computed Markov forward/backward recurrence values (s.markovF/s.markovB) for the segment under consideration

$$MP(s) = W_{dist} * mindist(s,P) + W_{heading} * \text{lw\_abs\_heading\_diff} + W_{roadclass} * abs(s.category - matchInfo.roadAttrs.category) + W_{markov} * (abs(s.markovF - matchInfo.markovF) + abs(s.markovB - matchInfo.markovB))$$

4. associate each ROI segment with its penalty value
5. in a Point decoding context, return the ROI segment s with the minimal penalty value, and its associated map-matched point oospp(s, P)
6. in a Linear decoding context, return an ordered candidate group containing the N consumer-side map segments having the smallest penalty values among all ROI segments (ordered in ascending order of penalty value), each with its associated map-matched point

Given a transmitted SIP P without attached matching information, in a context where map matching is intended, an analogous algorithm is applied, except that the penalty value will be defined by the segment's distance from P only:

$$MP(s) = mindist(s,P)$$

Map matching process for linear locations

Overall Process

The following steps are executed in order to identify the optimal consumer-side path corresponding to a sequence o SIPs with optionally-attached matching information:

1. For each transmitted segment identification point (SIP), with associated matching information if present, a group of the n best-matching candidate consumer-map segments is determined according to the previous subsection. In effect, a sequence of p ordered candidate segment groups [G_0, ..., G_{p-1}] is computed.
   Since the following algorithm works on edge granularity, an analogous sequence of

edge-level groups [EG_0, ..., EG_{p-1}] is determined by mapping each sequence of segment originating from the same edge to their corresponding edge.

2. Using the edge-level candidate groups the group multipath routing algorithm (see next subsection) computes, for all reachable edges e of EG_{p-1} a cost-optimal path with:
    a. The path ends in e.
    b. The path starts in an edge of EG_0 that is optimal with respect to e.
    c. For each intermediate group EG_i, the path involves some edge of EG_i, and traverses all groups in their natural order.
    An example is shown in Figure 2: Shortest Path determination between Edge-level Groups
3. Each of these paths is then subject to the following post-processing steps (cf. Figure 3: Postprocessing - adjusting path endings towards lowest penalty value :
4. On both ends of the path so obtained, the path is truncated by chopping off the outermost edge in case:
    - the outermost edge and the second-outermost edge both contain a candidate segment in the same (end) group, and
    - the candidate segment present on the outermost edge received a higher penalty value than the candidate segment present on the second-outermost edge in the individual segment matching step that shaped the candidate group.
    This process is repeated until no more edges can be pruned.
5. On both ends of the path, attempts are made to extend the path
    - by a hitherto-unused (within the path) edge from the same end group which is incident to the current end edge, has a lower penalty value than the current end segment, and offers a valid path extension according to its topological orientation vs. the path direction and the edge's oneWay property.
    This process, too, is repeated until no further extensions are possible.
6. The resulting edge-level result path is mapped back to segment level by mapping each edge to its corresponding segment list (see Figure 4: Mapping of edges to segments (only segments in end edges shown), pruning of endings to the optimal match points, selection of overall optimal path). On each end of the result path, if the junction flag is (present but) false, the respective end of the path is shortened (by chopping off outer segments) to the "best endsegment" – the segment having the lowest penalty value among all group segments which are elements of this end of the path.
7. The map-matched points are placed on the end segments: for a junction ending (indicated by the junction flag) on the respective junction point (i.e. one of the path's proper end points), otherwise on the optimum matching point of the respective (best) end segment.

Each of these segment-level paths is interpreted to extend on both ends only until the respective map-matched point. Taking this into consideration when comparing the path lengths, out of the small set of paths so obtained, the overall shortest path from EG_0 to EG_{p-1} is determined.

This is the best-matching segment-level contiguous path associated to the input SIP sequence, i.e. the result of the map-matching process (see Figure 4: Mapping of edges to segments (only segments in end edges shown), pruning of endings to the optimal match points, selection of overall optimal path).

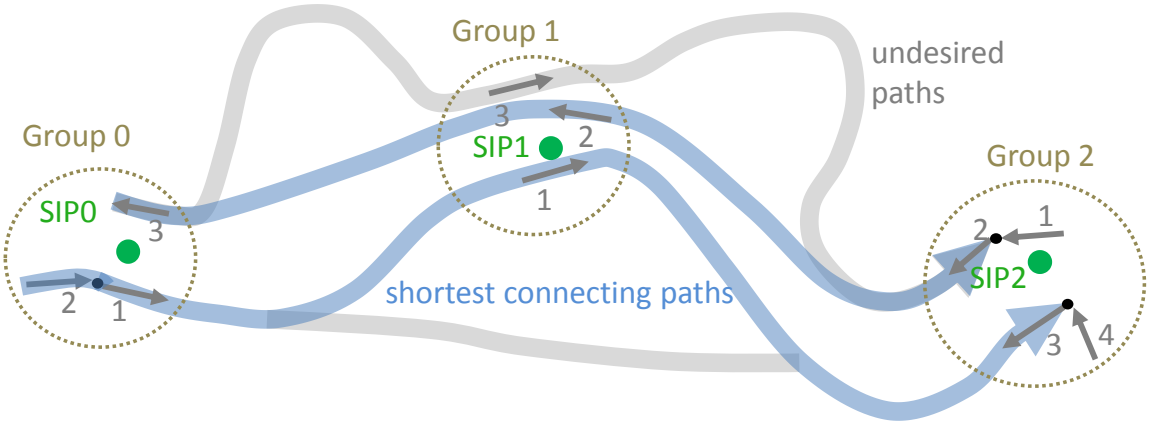

Figure 2: Shortest Path determination between Edge-level Groups

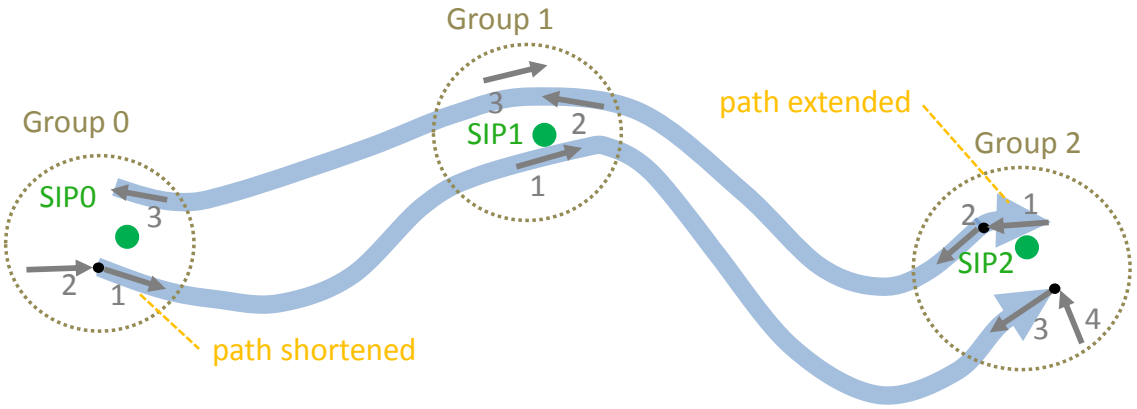

Figure 3: Postprocessing - adjusting path endings towards lowest penalty value

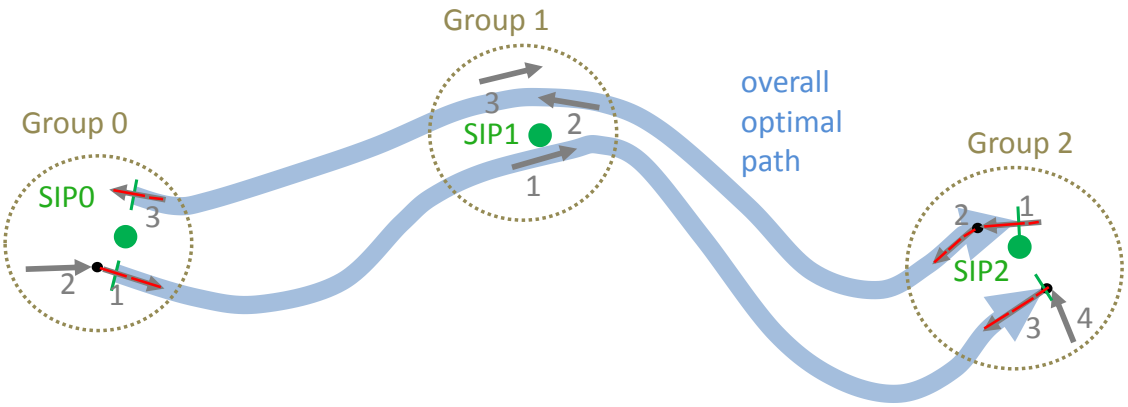

Figure 4: Mapping of edges to segments (only segments in end edges shown), pruning of endings to the optimal match points, selection of overall optimal path

Group Multipath Routing Algorithm

The objective of the algorithm presented here is to determine good edge-level approximations of the location path on the consumer-side map. This is done by determining all minimal, valid,

point-matched edge-level paths which connect all groups in ascending order. Of course only valid paths are considered, i.e. paths in which edges are being traversed against their topological orientation only if they don't have the oneWay property.

To enforce the latter constraint, and to have a more convenient basis for the routing algorithm, the representation for the map graph is modified now: The implicit one-way traversal constraints of the road network are made topologically explicit by transforming each edge that does not have the one-way property, in a "forward edge" and an anti-parallel "backward edge". The routing algorithm then operates exclusively on the edges of this graph. This edge-based routing paradigm enables more efficient and more flexible routing algorithms especially for road networks and navigation-specific requirements (such as accommodating turn costs). Therefore, it is already used in similar form in a considerable number of navigation applications, and the necessary data structures (in particular, for efficient access to the direct successors of an edge) are already present. Thus the ULR algorithm does not impose unreasonable new data requirements on these systems. (Also, a sufficiently equivalent algorithm for working with other representations could be devised where the need arises.)

For the purposes of ULR, this map representation and associated data structures need not be present for the entire on-board map, but only for an adequately-sized region of interest. This region must contain all candidate edges, and all edges that may be involved in shortest paths connecting them. As the ULR method is free to place additional SIPs on the producer side if paths would be too hard to identify on the consumer side, this criterion can be reduced to a simple distance threshold; the region can then be conceptually determined by simulating painting the polygon line connecting the centroids of all candidate groups with a broad circular brush, and including all edges which are wholly or partially covered by this figure.

The ULR Group Multipath Routing algorithm, shown in Python-like pseudocode on the next page, operates in phases, each one finding *all* shortest connecting paths between the current and the next candidate group, in a single run of a Dijkstra-like propagation method.

In each phase, all candidate segments of the current group which had been reachable in the previous phase are inserted into a priority queue, from which then "focus" segments repetitively are taken until the queue is empty. For each focus segment, all its direct successors are considered, checking whether a shorter path to the current successor than previously-known has been just found; if so, the new path and minimal cost is stored. If the current successor belongs to the next group, or prolongs the path too much, propagation stops right there, otherwise the successor edge is inserted into the priority queue again, so its own successors will be analyzed later as well.

For edges in the start and end group, non-standard costs must be considered and stored when the "forward" and "backward edges" are generated from the "bidirectional" (i.e. non-oneway) edges of the original road network, as follows:

position and role of edge	**Forward edge**	**Backward edge**
Start group edge	Post-match part length	Pre-match part length
End group edge	Pre-match part length	Post-match part length

This is because for edges at the beginning or end of the path, only the part before or after the match point will contribute to the actual path length eventually.

```
algorithm ULRGroupsMultipathRouting( AllGroups, forbiddenEdge=None ):
  # AllGroups = [ [edge01, edge02, edge03], ... ] - each group: list of candidate edges in ascending order of penalty

  edgeToPath = {}                    # map: candidate edge -> shortest path found from first group to this edge
  for edge in AllGroups[0]:          # initialize with trivial paths,
    edgeToPath[edge] = [edge]        # each containing just one start group edge

  for curGroupNo in range (0, len( AllGroups) - 2 ): # proceed in phases, each: shortest routes current->next group
        # (A) phase initialization
        for e in AllEdges :                        # all edges of considered map region
              minCost[e] = +INFINITY
              back[e] = None
        queue = PriorityQueue()                    # initialize priority queue
        currentGroupMaxCost = 0       # cost of longest of shortest paths to currentGroup (from previous phase)

        # (B) enqueue edges of current group that were reached in the previous phase
        for edge in AllGroups[curGroupNo]:
              partialPath = edgeToPath.get(edge)
              if partialPath is None:              # no path was found from first group to this edge
                   continue
              partialPathCost = getCost(partialPath)
              minCost[edge] = partialPathCost  # register best cost from earlier phases for the starting edges
              back[edge] = edge                    # end mark for backward path tracing after propagation phase
              queue.put(edge, minCost[edge])
              # also determine currentGroupMaxCost
              if partialPathCost > currentGroupMaxCost:
                    currentGroupMaxCost = partialPathCost

        # (C) propagation
        while not queue.empty():    # phase terminates when teher's nothing left in the queue
              focusEdge = queue.get()                  # consider first edge from queue
              for (succEdge in focus.Edge.successorEdges()   # for all direct successors of the focus edge
              if succEdge == forbiddenEdge:                # don't consider routes through this edge
                    continue
              double costWithSucc = minCost[focusEdge] + succEdge.cost
              if costWithSucc  >= minCost[ succEdge ]:
                    # less costly path from current group to succ already known
                    continue
              minCost[ succEdge ] = costWithSucc
              back[ succEdge ] = focusEdge          # shortest path goes over focusEdge->succEdge, store that
              if costWithSucc  > currentGroupMaxCost + ignoreThresholdDist( curGroupNo ):
                    # successor makes path "too long"  (longer than an ignore threshold, e.g. 3*estimated
                    # current/next group distance), thus no need to propagate beyond this successor
                    continue
              if succEdge in AllGroups[ curGroupNo+1 ]:  # reached next group, thus no need to propagate further
                    continue
              queue.put( succEdge, costWithSucc )       # enqueue succEdge so propagation will go beyond it
```


After the propagation process terminates, the found per-phase paths are reconstructed from the backpointer list, and concatenated to obtain the overall paths (this is not shown in the pseudocode), which connect all of the groups in ascending order.

The resulting edge-level paths are optimal w.r.t. the pure cost (i.e. path length) goal function, but not necessary yet w.r.t. the real objective of finding the best-matching path (because the optimally matching path might contain more than one edge from the same group). Therefore the post-processing steps described in the previous subsection still need to be applied.

The algorithm is also able to take an optional parameter forbiddenEdge into account, and will only identify paths not using that edge if the parameter is present. This feature enables the algorithm to be used for determining the characteristic edges of a linear location on the producer side (see next section).

Encoding a Linear Location

Depending on the application context, different types of producer-side original linear location information (producer-level information, PLI) may be available, all of which need to be usable for generating Linear ULR references. Thus several algorithms are presented which, depending on the input, need to be combined to transform it, in a stepwise manner, until the actual binary encoding can take place. The last step (binary encoding, referred to as Algorithm A in Figure 5), is omitted here for brevity. Instead the remainder of the text focuses on the high-level transformations (algorithms B and C).

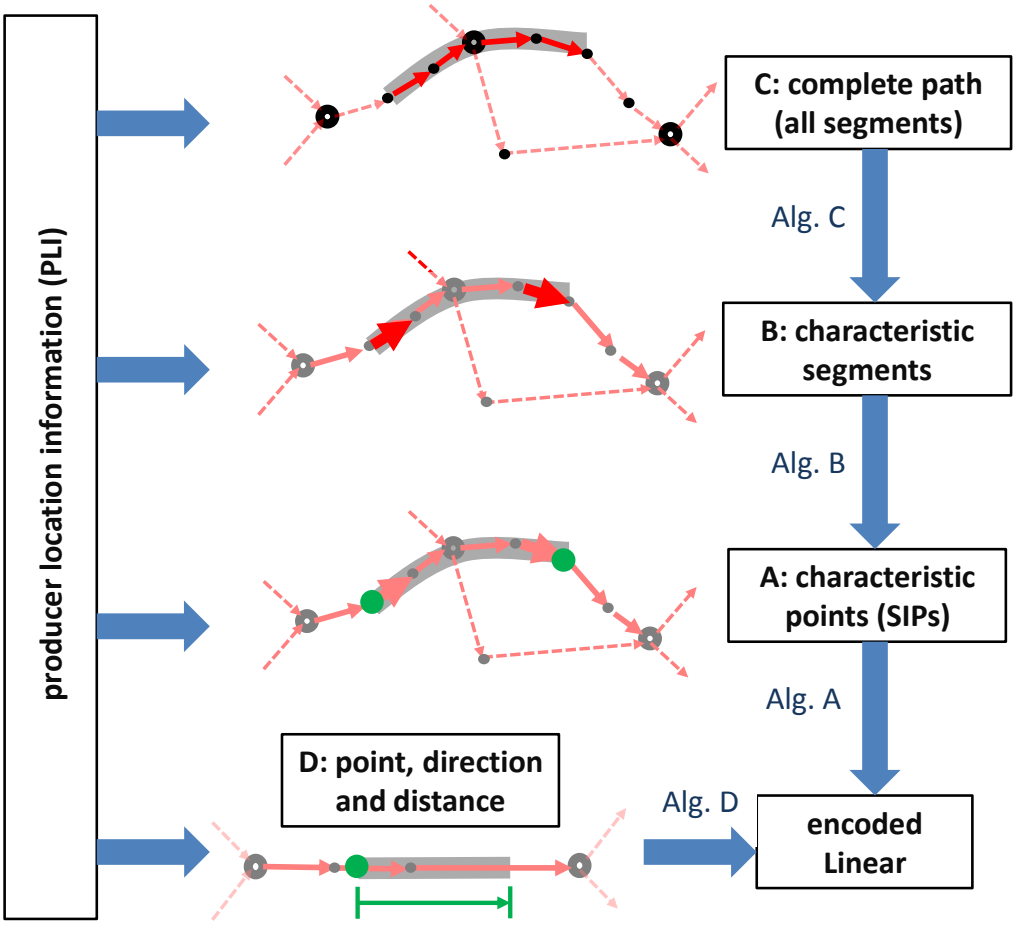

Figure 5: Creating Linear Locations from Different Inputs

Input B: PLI provides (a small number of) characteristic segments, i.e., an ordered list SEGS of characteristic producer map segments (or a list SEGS_SEQ of such lists).
The input segments shall be sufficient to characterize a contiguous, directed path (or sequence of connected sub-paths) on the producer-side map. Note that they are not assumed to actually *form* the contiguous path on the producer-side map. Instead, they shall indicate the beginning and end of the path, and (as far as needed) all important turns in between so that the entire path is uniquely *characterized* by them. See Algorithm C below for how to *select* characteristic segments in case the PLI only provides the *entire* path in the producer-side map.

Algorithm B:

1. if the input is a list of lists SEGS_SEQ, flatten it into a simple list (SEGS)
2. for each segment seg in SEGS, determine a SIP p as described below, associate seg as its segment reference to it, and collect p into a POINTS list
3. execute Algorithm A on POINTS.

   Each SIP p is placed on its respective segment as follows:
   a. for the first segment segFirst in SEGS,
      - if segFirst.FromNode is a junction node, p is placed on the interior of segFirst in a distance of about 20m from segFirst.fromNode[3] (see Figure 6). In case segFirst is shorter than this distance, segFirst.toNode is used as p.
      - else (i.e. segFirst.FromNode is a shape point), segFirst.FromNode is used as p.
   b. for each inner segment seg in SEGS, p is placed on the center point of seg
   c. for the last segment segLast in SEGS,
      - if segLast.toNode is a junction node, p is placed on the interior of segLast, in a distance of about 20m from segLast.toNode[4] . In case segLast is shorter than this distance, segLast.FromNode is used as p.
      - else (i.e. segLast.toNode is a shape point) segLast.ToNode is used as p

   Note: This automatic SIP placement algorithm is only used when no characteristic points are available from the PLI. (SIPs coming from the PLI will be left untouched.)

[3] This measure serves to reduce the risk of false segment matches on the wrong side of the junction node (the outer side w.r.t. the path). Of course the risk for false matches on the inner side will be increased, but such false matches are fixed during decoding, since in this situation the junctionFlag will be set as well.
[4] See previous footnote.

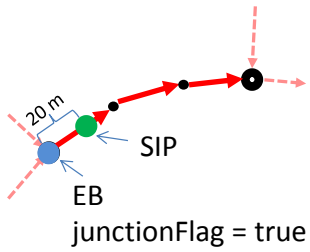

Figure 6: Event Begin is a junction[5]

Input C: PLI provides an entire, contiguous path on the producer-side map in the form of an ordered list of direction-attributed segments ALLSEGS (i. E. no characteristic points/segments have been selected yet).

Algorithm C:

The algorithm is presented as Python-like pseudocode below. It uses the multi-path routing algorithm presented in the previous section.

The idea is to trace the path forward from a focus edge (initially placed on the location path's start edge) towards the end of the location path and to assess, at each junction, the risk that an incorrect path connecting the focus edge with this junction might get favored on the consumer side (because the undesired partial path has an approximately equal or lower length than the corresponding part of the true location path). If this risk is high, the correct inbound edge of the junction is marked as characteristic (so a SIP enforcing its use on the consumer side will be generated). After this, the edge just marked is made the new focus edge, and the process is repeated until the end of the path is reached or no more alternatives paths are found. To make the method robust against path length deviations between producer and consumer map, alternative paths are considered a risk even if they are somewhat longer than the corresponding partial correct location path.

The result CHARSEGS is the input for Algorithm B above.

[5] This analogously applies for EventEnd on a junction.


```
algorithm segPathToCharEdges(segPath):
  SECURITYFACTOR = 1.6
  edgePath=segPath.toEdgeLevel()
  numEdges = len(edgePath)
  INNERCHAREDGES=[]
  segFirst = segPath[0]
  segLast = segPath[len(segPath)-1]
  focusEdgeIndex = 0
  while focusEdgeIndex + 2 < numEdges:
          # include lengths of next two edges
          distFocusToCurEnd  = edgePath[focusEdgeIndex].length()
          distFocusToCurEnd += edgePath[focusEdgeIndex+1].length()

          #find first segment to which routes from focusEdge are ambigous
          foundCharEdge = false
          for curEnd in range ( focusEdgeIndex+2, numEdges ):
                  distFocusToCurEnd += edgePath[curEnd].length()
                  #find routes from focus edge ...
                  gFocus = [edgePath[focusEdgeIndex]]
                  # ... to current end ...
                  gCurEnd= [edgePath[curEnd]]
                  groups = [gFocus, gCurEnd]
                  # ... NOT going through the location path edge preceding the current-end edge
                  criticalEdge = edgePath[curEnd-1]
                  altPaths = groupsMultipathRouting(groups, criticalEdge)
                  if not altPaths: # no alternative paths at all => no risk
                      continue
                  shortestAltPath = altPaths.sortByLength()[0]
                  if shortestAltPath.totalLength() > SECURITYFACTOR*distFocusToCurEnd:
                      # even shortest alternative path is above security threshold  => no risk
                      continue
                  # an alternative path shorter than the security threshold was found
                  # => mismatch risk , the critical edge must be marked characteristic
                  INNERCHAREDGES.append(criticalEdge)
                  focusEdgeIndex = curEnd-1 # in next (outer) iteration, continue from this edge
                  foundCharEdge = true
                  break
          if not foundCharEdge:
                  break # no characteristic edges between last focusEdge and end of path - finished
  CHARSEGS = []
  # create complete list of characteristic segments
  CHARSEGS.append(segFirst)  # first segment is always characteristic
  for edge : INNERCHAREDGES
          CHARSEGS.append(edge.selectSegment()) # select an inner segment of this edge
  CHARSEGS.append(segLast)  # last segment is always characteristic
```


Conclusion

This paper documents the central algorithms used to encode and decode linear locations in TPEG2-ULR. In contrast to other existing location referencing methods, the multi-path routing algorithm presented here enables a location referencing process that is very robust against map deviations, even deviations that reverse the path length ratios between (sub-paths of) the correct location path, and neighbouring, undesired alternative (sub)paths. Preliminary testing work has yielded very good matching quality.

References

[1] TPEG – Transport Protocol Experts Group; http://www.tisa.org/technologies/tpeg/
[2] TISA – Traveller Information Services Association; http://www.tisa.org/
[3] TMC – Traffic Message Channel; http://www.tisa.org/technologies/tmc/
[4] RDS-TMC (ISO 14819 Series); http://www.iso.org/
[5] TPEG Binary Specification – Generation 1 (CEN/ISO TS 18234 Series)
[6] tpegML (XML) Specification (CEN/ISO TS 24530 Series)
[7] TPEG2 – Generation 2 (prCEN ISO/TS 21219 Series)
[8] Schramm, A.; Kwella, B.; Schmidt, M.; Pieth, N.; Ernst, T.: Universal Location Referencing: A new Approach for dynamic Location Referencing in TPEG. Progress Report, TPEG-LOC2 Study,. Fraunhofer Publica, 2012. http://publica.fraunhofer.de/dokumente/N-192560.html
[9] Schmidt, M.; Schramm A.: TPEG-ULR: A new Approach for Dynamic Location Referencing In: Proceedings 19th ITS World Congress 2012, Oct. 22-26, 2012, Vienna, Austria.

